

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

Assignment - 1

Student Name: Anish Yadav

Branch: BE- CSE

Semester: 6th

Subject Name: System Design

Subject Code: 23CSH-314

UID: 23BCS13849

Section/Group: 23BCS_KRG_1A

Date of Performance: 04/02/26

Q1. Explain the role of interfaces and enums in software design with proper examples.

Solution-

Interfaces in Software Design

An **interface** in object-oriented programming is a reference type that defines a set of abstract methods. It specifies **what a class must do**, but not **how it should do it**. Interfaces are fundamental in building scalable and maintainable systems.

Roles of Interfaces

1. **Abstraction** - Interfaces hide implementation details and expose only behavior.
2. **Loose Coupling** - Systems depend on contracts rather than concrete implementations.
3. **Polymorphism** - Multiple classes can implement the same interface differently.
4. **Extensibility** - New implementations can be added without modifying existing code.

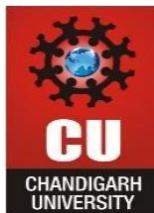
Example of Interface Usage

```
// Interface
public interface PaymentMethod {
    void pay(double amount);
}

// Implementation 1
public class CreditCardPayment implements PaymentMethod {
    @Override
    public void pay(double amount) {
        System.out.println("Payment made using Credit Card: " + amount);
    }
}

// Implementation 2
public class UpiPayment implements PaymentMethod {
    @Override
    public void pay(double amount) {
        System.out.println("Payment made using UPI: " + amount);
    }
}

// Service class using interface
public class PaymentService {
    private PaymentMethod paymentMethod;
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
public PaymentService(PaymentMethod paymentMethod) {  
    this.paymentMethod = paymentMethod;  
}  
  
public void processPayment(double amount) {  
    paymentMethod.pay(amount);  
}
```

Here, **PaymentService** works with any payment method without knowing its internal implementation.

Enums in Software Design

An **enum (enumeration)** is a special data type used to define a collection of **fixed constants**. Enums enhance code safety, readability, and maintainability.

Roles of Enums

1. **Type Safety** – Restrict values to predefined constants
2. **Improved Readability** – Replaces ambiguous numbers/strings
3. **Maintainability** – Centralized definition of allowed values
4. **Better Control Flow** – Works efficiently with switch statements

Example of Enum Usage

```
public enum OrderStatus {  
    PLACED,  
    SHIPPED,  
    DELIVERED,  
    CANCELLED  
}  
  
public class Order {  
    private OrderStatus status;  
  
    public void setStatus(OrderStatus status) {  
        this.status = status;  
    }  
  
    public OrderStatus getStatus() {  
        return status;  
    }  
}
```

Using enums prevents invalid values:

```
order.setStatus(OrderStatus.SHIPPED); // Correct  
order.setStatus(null); // Avoidable error
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

Q2. Discuss how interfaces enable loose coupling with example.

Solution-

Loose Coupling

Loose coupling refers to a design approach where system components have minimal dependencies on each other. Changes in one module do not significantly impact others.

Tightly Coupled Design (Without Interface)

```
public class EmailService {  
    public void sendEmail(String message) {  
        System.out.println("Email sent: " + message);  
    }  
}  
  
public class Notification {  
    private EmailService emailService = new EmailService();  
  
    public void notifyUser(String message) {  
        emailService.sendEmail(message);  
    }  
}
```

Issue

The **Notification** class is directly dependent on **EmailService**. If we want to switch to SMS or push notifications, we must modify the class.

Loosely Coupled Design (Using Interface)

```
public interface MessageService {  
    void sendMessage(String message);  
}  
  
public class EmailService implements MessageService {  
    public void sendMessage(String message) {  
        System.out.println("Email sent: " + message);  
    }  
}  
  
public class SmsService implements MessageService {  
    public void sendMessage(String message) {  
        System.out.println("SMS sent: " + message);  
    }  
}  
  
public class Notification {  
    private MessageService messageService;  
  
    public Notification(MessageService messageService) {  
        this.messageService = messageService;  
    }  
    public void notifyUser(String message) {  
        messageService.sendMessage(message);  
    }  
}
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

Benefits of Loose Coupling Through Interfaces

Aspect	Explanation
Flexibility	Easily switch between Email, SMS, etc.
Testability	Mock implementations can be used
Maintainability	No changes needed in Notification class
Scalability	New message services can be added easily

Q3. Design an HLD for a Payment Processing System, showing where interfaces would be used.

Solution - HLD Diagram [draw.io](#) file is attached with this assignment PDF.